

Driving Innovation with Technology:

THE INTELLIGENT
USE OF OBJECTS



Taligent™

DRIVING INNOVATION WITH TECHNOLOGY:

THE INTELLIGENT USE OF OBJECTS

A TALIGENT TECHNOLOGY
WHITE PAPER

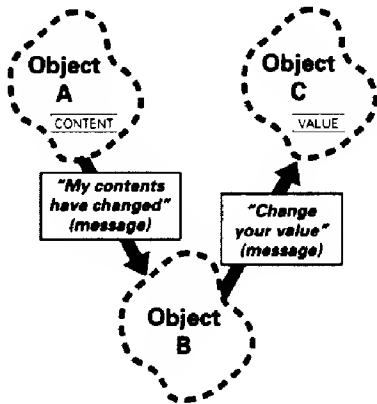
Table of Contents

<i>Executive Summary</i>	3
<i>Today's Software Crisis</i>	4
<i>Software Development: A Historical Perspective</i>	4
<i>The Taligent Solution: Objects-From-The-Bottom-Up</i>	6
<i>Basic Object Concepts</i>	8
<i>Major Advantages of Frameworks</i>	12
<i>How To Get Started</i>	13

Executive Summary

Object-oriented technology (OOT) is earning its place as one of the most important new technologies in software development. It has already begun to prove its ability to create significant increases in programmer productivity and in program maintainability. By engendering an environment in which data and the procedures that operate on the data are combined into packages called objects and by adopting a rule that demands that objects communicate with one another only through well-defined messaging paths, OOT removes much of the complexity of traditional programming (see Figure 1).

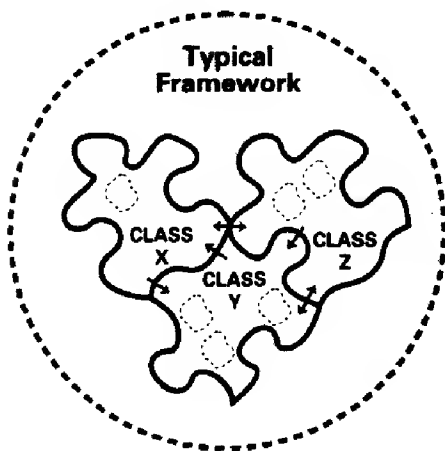
Fig. 1



Taligent Inc., an independent joint venture of Apple Computer, Inc. and IBM Corporation, is realizing the full promise of OOT through the use of frameworks. A framework defines the behavior of a collection of

objects and represents a way to reuse high-level designs (see Figure 2). Taligent is taking the use of frameworks to its logical conclusion

Fig. 2



An abstract class is a design for a single object. A framework is the design of a set of objects that collaborate to carry out a set of responsibilities. •

by extending these frameworks down into the domain of system software, up to the level of immensely powerful application components, and out to the development environment itself. The result

is an exciting new technology foundation that will accelerate the pace of innovation by systems manufacturers, software developers, and all types of solution providers.

This paper describes, at a high level, Taligent's philosophy and approach to designing its open and extensible system software platform. It is the first in a series of such papers that will provide an increasingly focused view of Taligent's new operating environment. Written primarily for software developers and designers, including independent commercial vendors and in-house corporate programmers, this paper also addresses issues of concern to hardware designers and end-users of applications. Technical managers, strategic planners, and others interested in the future of software development will also find this paper useful. It assumes a rudimentary understanding of objects and of object-oriented programming (OOP) basics, but no prior knowledge of a specific programming language and design methodology or of the relatively new concept of frameworks is assumed.

The result is an exciting new technology foundation that will accelerate the pace of innovation by systems manufacturers, software developers, and all types of solution providers.

Driving Innovation with Technology: The Intelligent Use of Objects begins with a brief historical overview of microcomputer software development. This description is designed to put Taligent's technology approach into perspective both in terms of the problems it solves and of the roots from which it grows. Building on that perspective, the next section addresses the advantages and benefits of frameworks as seen from the viewpoint of software developers and hardware designers. The final section defines frameworks in greater detail, offers some insights into their importance, and describes some sample hypothetical frameworks and their potential use.

Today's Software Crisis

Hardly anyone would disagree with the observation that the software industry is in the midst of a crisis. Some of the key characteristics of this crisis include:

- New commercial applications take more than two years to bring to market.
- Software projects require too many engineers; consequently, the projects are expensive and difficult to manage.
- New development has slowed dramatically because of the complexity in delivering the baseline function expected by users and the lack of tools to create interoperating and custom solutions easily.
- In large organizations, maintenance of existing code still occupies far too large a portion of the development group's time and resources.
- Product quality is static or worsening, despite the increased emphasis on testing.

As a direct result of this crisis, hardware and software innovation is stagnating at a time when users' expectations for their systems are increasing. Hardware vendors can't create radically new designs because the lag between the completion of their designs and the availability of system-level software to support those designs is unacceptably long. Software developers who wish to innovate find themselves in one of two equally unacceptable situations. Either they are in organizations that are small enough to avoid the development bottleneck problems but too small to have marketing impact, or they are in big organizations that could bring significant marketing clout to bear on products, but those products never seem to be completed on time.

How did the industry reach this point? Why is an industry that should be characterized by easy adaptation to rapid change suddenly stuck? Taligent believes there are sound historical reasons for this situation and that a discussion of these reasons will help identify the right solutions to the crisis the software industry now faces.

Software Development: A Historical Perspective

The history of desktop computer programming presents a pattern of peaks and valleys. As enterprising companies invent new tools and paradigms to help programmers simplify the difficult aspects of programming, other enterprising companies are busy building on new hardware developments. The latter companies, taking advantage of exponentially increasing rates of change in hardware power, pack more capability and expectations into the desktop computer. This, in turn, creates a new round of difficulties for software developers, spawning yet another group of enterprising software tool companies.

Table 1 summarizes the history of the first three decades of personal computer software development.

Table 1. FIXING WHAT'S HARD

Decade	What's Hard
1970s	Everything
1980s	GUI
1990s	Program functions and interoperability

This process began in the 1970s. When desktop computers began to realize their first glimmer of commercial success, literally everything about programming them was hard. The first programmers of personal computer software needed intimate knowledge of the hardware. There were no standard operating systems to speak of. Designers wrote their own operating systems and typically used low level tools.

Literally as soon as these first computers began to appear, some programmers began to tackle the difficulties by thinking about the kinds of tools they would like. First, program assemblers appeared. Later, the first high-level language, BASIC, became available. As increasingly powerful and sophisticated programming languages and tools emerged, details of hardware operations were one or two steps removed from the programmer.

The 1980s: GUI Gains Focus as “The Hard Part”

In 1984, the rules changed dramatically. Apple Computer introduced the Macintosh and the ante went up several notches. The Macintosh was to become successful particularly because of its amazingly easy-to-use interface. It was unleashed however on an unsuspecting world of programmers who were soon to discover the bitter truth of the aphorism, “Easy to use is easy to say.” Suddenly, the really hard part of developing software wasn’t the algorithms that solved the user’s underlying problem but the creation of the user interface and the management of an event-driven system. The old tools were of no help in addressing these problems.

True to its still-brief history, the industry quickly spawned a host of tool builders to tackle these difficulties. Programming languages were extended and program skeletons developed.

As the decade of the 80s drew to a close, OOP exploded from its relative obscurity. Several system software companies — notably Apple Computer and Next, Inc. — quickly saw the advantages of this new technique and created programming tools that provided the skeleton of a finished application in OOP form. Meanwhile, the rest of the cycle continued. Hardware designers continued to build new components and other enhancements. Users became comfortable with their graphical interface components. More importantly, they also became more comfortable with the notion that programs need not work in isolation from one another. Achieving this interoperability of software across dissimilar platforms and across networks continues to be a major challenge facing software developers of the 1990s.

The 1990s: Mastering Complexity is the Challenge

For the most part, software developers are trying to face the immense challenges of the 1990s with tools that are not nearly as sophisticated as the products they try to build with

those tools. Object-oriented languages are rapidly becoming widely adopted and their use has had a significant positive impact in many areas of software development. Many of its advantages arise from the degree to which object-oriented designs and programs tend to be implementation-independent and thus highly reusable. Still, the reuse level of program code is below what most of OOP’s strongest advocates have long believed should be attainable. The development environments in which these tools are used are, for the most part, last-generation designs. For example, debuggers often aren’t designed to deal with objects.

Software developers are trying to face the immense challenges of the 1990s with tools that are not nearly as sophisticated as the products they try to build with those tools.

Even the availability of class libraries — collections of pre-built objects designed for reuse — hasn’t been a complete answer to the needs of the modern software developer. The dream of programming by connecting objects together and writing a small amount of code specific to a given application remains elusive.

In part, Taligent believes, the problem is that the objects in these libraries as well as those created by even the most accomplished and experienced object programmers, are simply not sufficiently high-level. They tend to become specialized, rather than generic and reusable. They leave far too much of the complexity of modern software development in a place where the programmer must be aware of it and deal with it, often with little or no tool support.

Still another problem with today’s object libraries lies in their very size and complexity. As class libraries have been built to address complex software requirements, these libraries themselves have become interconnected and complex. Accomplished OOP programmers have been known to spend more than half their time “shopping for code,” i.e. searching through their class libraries to find the right starting point for a particular project.

Most, if not all, existing class libraries were built around the assumption that a completely generic, reusable set of code was essential to the success of OOP. Thus while individual objects might tend to become quite specialized and lose some of their reusability in the process, the aggregation of these objects tended to grow in unmanageable and even unpredictable ways in an effort to make the whole library more generic. The result was too complex to understand and use.

The classes and methods involved in text processing, for example, might be spread over a large part of a class library. Creating even a relatively simple text editor can become a major challenge. Writing a full-blown word processor in such environments presents huge obstacles. If that word processor must not only be as powerful as any that are commercially available, but also flexible and adaptable and user-programmable, the job becomes formidable indeed.

The Taligent Solution: Objects-From-The-Bottom-Up

Taligent takes the position that providing the tools developers need to tackle the complexity of the 1990s software world requires two critical components: a well-designed library of extremely high-level abstractions of objects that take the application development process as far as possible integrated into the operating system; and a set of development tools that is designed with OOP in mind from the beginning. This strategy has three major elements:

1. Combining objects into highly reusable and richly expressive frameworks.
2. Extending these frameworks into the system services level of the environment.
3. Creating an object-oriented development environment specifically and intimately connected to the framework approach to software design.

What are Frameworks?

Taligent believes that frameworks, which are central to the design of its operating environment, are the most important advance in object design.

If objects are abstractions that are sometimes difficult to explain, frameworks present an even more intriguing challenge. One of the most cogent definitions of frameworks comes from Ralph E. Johnson of the University of Illinois and Vincent F. Russo of Purdue. In their 1991 paper, *Reusing Object-Oriented Designs*,[•] they offer the following explanation:

“An abstract class is a design for a single object. A framework is the design of a *set* of objects that collaborate to carry out a set of responsibilities. Thus, frameworks are larger scale designs than abstract classes. Frameworks are a way to reuse...high-level design.”

A key distinction between a framework and an arbitrary collection of classes, however closely related those classes might be functionally, is that a framework describes not only the objects but also their interactions with one another.

The word *framework* has also sometimes been used in OOP circles to describe a very high-level abstraction: a program template. In that context, a framework provides a complete application; the developer “simply” adds the functionality specific to a particular program. (MacApp™ from Apple Computer, Inc. and OWL™ from Borland International are such frameworks.)

By applying the very techniques that make object programming useful to the program building blocks themselves, frameworks remove much of the complexity of today's object-based systems.

There is really a continuum of frameworks, from very simple ones to very powerful ones. More benefits are derived from the larger-scale frameworks than from the simple ones. However, designing the larger scale frameworks require more OO design expertise

[•] Ibid.

as well as the realization that OOP can be applied at a higher level. By applying the very techniques that make object programming useful to the program building blocks themselves, frameworks remove much of the complexity of today's object-based systems.

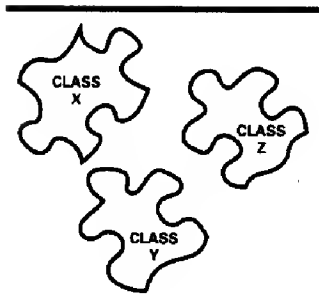


Fig. 3a

which the objects have been pre-assembled and their communications with one another clearly defined.

Fig. 3b

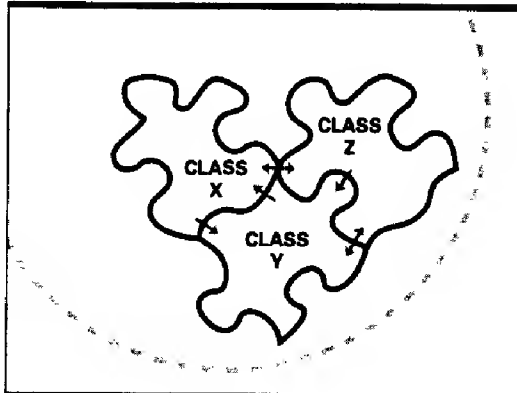


Figure 3 depicts the difference between a collection of objects (3a) and a framework (3b). The collection of objects (3a) can all be put together to create the framework, but the programmer has to do the work. Figure 3b symbolizes a framework in

Application-Level Frameworks

The basic premise of the Taligent operating environment from its earliest design (begun at Apple in 1988) was to extend both the reach and the scope of a MacApple-style framework to empower programmers beyond the level of the GUI.

It is premature to discuss any specific frameworks that might be included in Taligent's operating environment when it is complete. What follows is a discussion of the types of frameworks that would be possible to include in such a design.

Graphical editing is a function performed by all modern desktop computer systems on one level or another, so imagining a graphical

editing framework (GEF) is relatively easy. A GEF would provide a high level of flexible support for drawing, rotating, dragging, and scaling graphical objects. A GEF might include an understanding of both two- and three-dimensional graphic shapes and objects. By making this framework resolution- and device-independent, the developers of GEF could create a highly reusable, generic environment. The programmer who wants to build a graphical application on top of GEF, then starts with a very high degree of built-in functionality. What is there for this programmer to do? Freed from the necessity of being concerned with any of the functionality supplied by GEF, this programmer might focus attention on adding a light source for realistic 3-D appearance, combining predefined shapes into useful objects for a particular application, or collision detection for objects in a simulated environment.

To add these kinds of capability to the GEF, the programmer would use the framework's publicly defined interfaces, hooking into the many openings left in GEF by its designers. Figure 4 (page 10) shows one way of thinking about this process. The framework supplies a major part of the puzzle; the programmer creates a well-defined component that simply plugs into the puzzle and completes it as a finished application.

As an additional example, it might be interesting to think about creating a new framework for a future kind of technology.

Virtual reality (VR), as embodied in the science fiction of William Gibson, is viewed by many as having major potential impact on computers of the future.

A team of developers might well decide to create a Virtual World Framework (VWF) to provide a basis for other programmers to create virtual worlds and VR-based products of many kinds. Such a framework might include such things as:

- plot outlines for the interaction of the user with objects in the virtual reality

Basic Object Concepts

To understand what makes frameworks significantly more powerful than objects and classes, it is important to understand some basic object technology concepts.

Even if the reader is comfortable with OOP and its basic concepts, this section will be helpful in understanding the focus of this paper. The usefulness of this section stems in part from the confusion that exists in the development arena around the key ideas in OOP. In some sense, "object-oriented" has become a buzzword. As such, it has enjoyed the fate of most buzzwords in this industry: it has been pre-empted and corrupted by many products claiming to be object-oriented only so they could sell more copies. The major concepts involved in OOP are encapsulation, data abstraction, inheritance, and polymorphism. These terms are explained in detail in the remaining paragraphs of this section.

Data Abstraction

The fact that objects are encapsulated produces another important fringe benefit that is sometimes referred to as *data abstraction*. Abstraction is the process by which complex ideas and structures are made more understandable by the removal of detail and the generalization of their behavior. From a software perspective, abstraction is in many ways the antithesis of hard-coding: if every detail of every window that appears on the user's screen in a GUI-based program had to have all of its state and behavior hard-coded into a program, both the program and the windows it contains would lose almost all of their flexibility. By abstracting the concept of a window into a window object, object systems permit the programmer to think only about the specific aspects that make a particular window unique. Behavior shared by all windows, such as the ability to be dragged and moved, can be shared by all window objects.

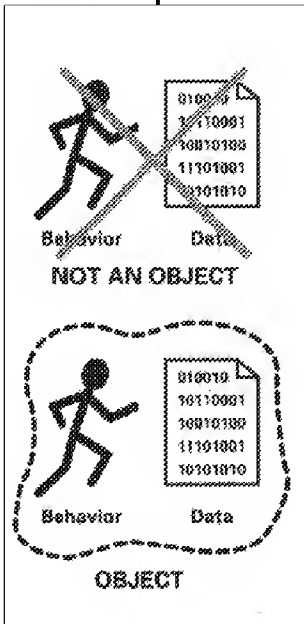
Encapsulation

Objects are software components which combine data and the operations to be performed on that data into a single entity. This feature of objects is called *encapsulation*. In that sense, a window is not an object merely by the fact of its existence as something people often think of as an object in their real-world experience. Perhaps the greatest single benefit of encapsulation is the fact that the state of any object can only be changed by well-defined methods associated with that object. When behavior is confined to such well-defined locations and interfaces, changes in


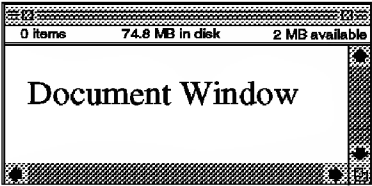
them will have minimal impact on the other objects and elements in the system. A second "fringe benefit" of good encapsulation in object-oriented design and programming is that the resulting code is more modular and maintainable than code written using more traditional techniques.

Inheritance

As a natural outgrowth of encapsulation and abstraction, object-oriented development tools can and must include class libraries which feature *inheritance*. This is the mechanism by which abstractions are made increasingly concrete as *subclasses* are created for greater levels of specialization. For example, a class called Window might represent a rectangular area of the screen. Such a window might contain four points, one for each corner. Its behaviors might include such things as creating itself as a data structure, drawing itself on the screen, determining its height and width and calculating its area. Another class called DocumentWindow might inherit from the Window class so that it exhibits a rectangular shape made up of four points and includes the other functionality for a Window. But a DrawingWindow specializes the Window class by adding some display elements (a title bar, close box, scroll bars, and so forth). It might also add some behaviors, including such things as displaying its bitmap contents, resizing in response to the user's actions, etc.



Basic Object Concepts

CLASS	BEHAVIOR	
	<ul style="list-style-type: none"><input type="checkbox"/> Create<input type="checkbox"/> Calculate area<input type="checkbox"/> Draw<input type="checkbox"/> Determine height/width	
	<ul style="list-style-type: none"><input type="checkbox"/> Create<input type="checkbox"/> Calculate area<input type="checkbox"/> <i>Display contents</i><input type="checkbox"/> Determine height/width	<ul style="list-style-type: none"><input type="checkbox"/> <i>Resize</i><input type="checkbox"/> <i>Scroll</i><input type="checkbox"/> <i>Display bitmaps</i>

Without inheritance, this Window-Drawing Window hierarchy would require that both types of objects create their own rectangular areas on the screen, and that they write all of the methods associated with manipulating rectangles.

With programming languages like C++, a programmer can define a class which inherits from more than one parent class. This *multiple inheritance* feature adds a measure of power to OOP designs, but it also increases complexity. In a single-inheritance design, each class has one parent, leading to a tree structure that can be understood and traversed easily. Multiple inheritance designs create networks or webs of classes which are more difficult both to visualize and to traverse. In addition, protocols must be found for resolving name conflicts in traits inherited from two or more parent classes which have properties with identical names. However, multiple inheritance has proven useful in representing numerous valuable abstractions.

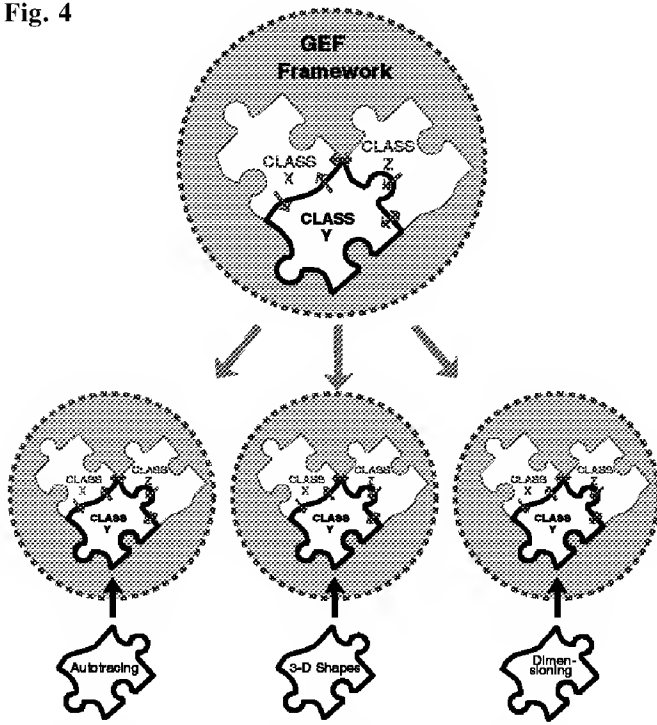
Inheritance is the main contributor to the increased programmer efficiency for which OOP is generally adopted. Inheritance makes it possible for developers to minimize the amount of new code they have to write to create applications. By providing a significant portion of the functionality needed for a particular task, classes in the inheritance hierarchy give the programmer a head start in program design and creation.

Polymorphism

One of the potential drawbacks to an object-oriented environment lies in the proliferation of objects that must exhibit behavior which is similar and which one would like to use a single message name to describe. For example, if a Draw message is sent to a Rectangle object, that object responds by drawing a shape with four sides. A Triangle object, on the other hand, draws a shape with three sides. Ideally, we want the object that sends the Draw message to remain blissfully unaware of either the type of object to which the message is addressed or of how that object will draw itself in response. If we can attain this ideal, we can then easily add a new kind of shape later (for example, a hexagon) and leave the code sending the Draw message completely unchanged.

In conventional languages, such a linguistic approach would wreak havoc. In OOT environments, the concept of *polymorphism* enables this to be done with impunity. As one consequence, methods can be written that generically tell other objects to do something without requiring the sending object to have any knowledge at all about the way the receiving object will understand the message. •

Fig. 4



The framework supplies a major part of the puzzle; the programmer creates a well-defined component that simply plugs into that puzzle and completes it as a finished application.

- sensor objects to facilitate the virtual world builder's ability to determine what happens when various kinds of objects encounter other objects or beings in the virtual world
- multiprocessing objects to permit changes in the VR to take place "off stage," making the user's experience feel more real-time than would be possible if changes always had to be made on the fly.

Frameworks encapsulate a group of closely related classes and make it possible for the programmer to work at a higher conceptual level of the design, being concerned only with interactions among object types with which he specifically needs to work.

Picture a programmer who wishes to create a virtual reality product to help students learn principles of designing urban housing structures as part of a municipal management program. Starting with the VWF, this programmer could quickly build up some objects

representing conceptual potential occupants of such housing as well as alternative layouts and approaches, and create a virtual world in relatively short order.

From these two hypothetical framework examples, one can appreciate the value and power of this natural extensibility. In today's object-oriented environments, these capabilities could be developed but they would consist of dozens, perhaps hundreds, of separate classes. The user would be faced with a staggering array of potential objects with which to interact and would have to learn the protocols for all of them.

Frameworks encapsulate a group of closely related classes and make it possible for the programmer to work at a higher conceptual level of the design, being concerned only with interactions among object types with which he specifically needs to work. The rest of the framework can simply sit politely and transparently in the background supporting the programmer's work.

System-Level Frameworks

Aside from the use of frameworks as building blocks, one of the most distinguishing characteristics of Taligent's operating environment is the totality of the use of objects and frameworks throughout its design. In fact, Taligent's differentiator from other system software platforms is its objects-from-the-bottom-up approach. Not only are the functions one generally thinks of encapsulating into objects contained in frameworks in Taligent's product, but system-level functionality is also defined in frameworks. This has some interesting consequences.

Many kinds of applications depend on intimate interaction with the system. For example, an electronic mail package will work better from the user's perspective if the program can access the desktop user interface. This would enable the application to post notification messages when mail arrives for the user, for example. In today's systems, where the system-level software is not open to such interaction, the designers of programs like an email package run into barriers.

They must either work around the wall — tunnel under it, so to speak — or let their applications work less than optimally.

By applying frameworks at the system level, Taligent opens a door in the walls that would otherwise pose significant obstacles for programmers. Taligent's research reveals that as much as half the code in many applications involves fixing or working around problems and obstacles posed by the operating system. In the Taligent operating environment, deliberate doors have been designed in from the beginning. This will significantly reduce the amount of time designers spend figuring out how to work around the deliberate walls in closed operating system interfaces.

Taligent's differentiator from other system software platforms is its objects-from-the-bottom-up approach.

System-level frameworks also avoid other common problems. For example, when new versions of hard-to-extend operating systems appear, developers of utility programs that take advantage of those systems or attempt to fill gaps in the systems' design often find their products break. Users who have come to rely on those utilities must either wait for the utility developers to catch up with the new release or work without their favorite utilities for some time. System level frameworks allow developers to revise their applications at a more rapid pace; one that is consistent and in sync with the revisions and extensions that are made to the system software.

People who write device drivers and other low-level software tools will find system frameworks to be supportive. Today, such programmers have no software interface to which they can write their routines; they are programming "at the metal," interacting directly with the hardware and the operating system.

Hardware developers gain a potentially powerful edge from the Taligent decision to implement frameworks all the way to the bottom of

the system. Because of the highly modular design the framework approach creates, it is possible to augment one or more classes with hardware components.

As an example of the value of the utility of frameworks in the hardware arena, think about the Virtual World Framework of the previous section. The programmer who built the interactive urban housing model has done so presuming a certain kind of interaction between the user and the system. Perhaps he assumed a traditional pointing device and a flat-screen display capable of realistic 3D graphics.

Two years after the program appears, the Virtual Life Company comes up with a new kind of glove, an enhancement of today's useful but expensive products. The glove can be sold for \$25, permitting tens of thousands of people interested in this kind of VR to get their hands on an intriguing new interface device.

The glove manufacturer creates an interface for the user by creating a system-level component that interacts with a user interface framework. Since the developer of the urban housing model product used the VWF and since the system is designed so that the VWF takes advantage of the user interface framework, the two-year-old application takes advantage of this new interface device with no program modifications.

By applying frameworks at the system level, Taligent opens a door in the walls that would otherwise pose significant obstacles for programmers.

This automatic inheritance of capability in frameworks means that users get the advantages of new technology immediately as it becomes available, not when all of the developers of all their programs are finally able to allocate the resources to adapt their programs.

Development Environment-Level Frameworks

From the outset, Taligent felt it was important to extend the framework approach to the development environment as well. This allows the same flexibility and customizability as described at other levels of the operating environment. And, since it is built around the notion of frameworks, it is, naturally, open to easy modification and extension. Third-parties can develop complimentary products, such as language-sensitive editors, debuggers, browsers and similar tools that can be easily adapted to the existing development framework.

In addition, Taligent recognized that if frameworks are going to be useful, they cannot be hard for developers to use. The company has placed as much emphasis on designing and creating development tools as they have on developing frameworks, knowing the big issue with development frameworks, knowing the big issue with development tools and languages is not so much what one can do as it is how productively one can do it.

Major Advantages of Frameworks

If, as Taligent contends, mastering complexity is the important challenge facing software developers in the 1990s, why hasn't object technology lived up to its advance billing and solved this problem? Two of the primary reasons were pointed out earlier when describing the complexity challenge. First, the class libraries associated with OOP languages tend to abstract functionality at a low level. However, low-level abstractions don't fully realize the benefits of OOP.

Second, these collections of classes tend to create their own layer of complexity. In the same respect, Taligent believes data abstraction to be "a double-edged sword," recognizing that without abstraction, complexity can't be dealt with at all. Once concepts are abstracted into classes, the abstraction creates its own new kind of

complexity. The complexity of the problem and of the underlying design elements has to be managed, since everything has been abstracted.

Frameworks address these two problems by bringing higher level abstractions to bear on the design of the libraries themselves. This results in higher-level design elements, which reduces greatly the two-tiered complexity problem. In many ways, this use of objects to create higher levels of abstraction (frameworks) is similar to using a language to write a compiler for that language.

If most programmers now spend more than half their time working around or overcoming problems in the operating system, the application of frameworks to that level of software alone greatly reduces programming requirements.

Another advantage of the framework approach to OO development is the leverage the developer gains. If most programmers now spend more than half their time working around or overcoming problems in the operating system, the application of frameworks to that level of software alone greatly reduces programming requirements. Beyond that, the reuse of design that is the promise of objects is achieved on a much higher level with frameworks. The capabilities included in any framework automatically accrue to any program that simply makes use of that framework. And, frameworks provide built-in functionality that is still customizable at a fine level.

Beyond productivity gains, frameworks have another significant advantage. Taligent's new operating environment will provide so much functionality in its frameworks that software designers will be able to contemplate and create whole new kinds of software. This extensive, built-in functionality will eliminate much of the mundane work that programmers have to do today to get a program working. Instead, programmers will be able to focus on developing the unique aspects of their programs.

Another major advantage of a smoothly integrated library of frameworks is that by their nature they encourage and facilitate cooperation and interoperability among applications. When all of the underlying assumptions and models used by most applications are similar or identical, getting those programs to talk to one another and share data as well as control becomes a much more achievable objective.

The reuse of design that is the promise of objects is achieved on a much higher level with frameworks.

Getting the software development community from where it is today to where Taligent believes it must eventually be, will not happen overnight. Developers who see the potential in frameworks to make their software more powerful and their business opportunities broader might well be interested in what they can do today, before the Taligent operating environment is available, to get ready for this next wave of technology.

How To Get Started

Taligent suggests developers interested in developing for the Taligent operating environment take the following early steps:

1. Learn object-oriented design: Developers who simply use C++ as a better C without fundamentally changing their design approach to use encapsulation, polymorphism and inheritance will not realize the substantial benefits of object-oriented technology. There are several books and commercially available training courses to help learn OOD.
2. Learn an object-oriented language and begin to use it exclusively. C++ is the first language that will be supported in the Taligent operating environment, but it is expected that other OO languages will support the system.
3. Learn to design and work with frameworks. Begin with class libraries and understand both their power and their limitations. Then begin to think in terms of frameworks. Becoming familiar with commercially available frameworks such as MacApp and OWL will help the learning process.
4. Imagine and design new kinds of software that would be possible in the context of powerful framework-based systems and begin to shape designs toward that eventuality.
5. Call Taligent's Developer Partnerships voice mailbox at 408•777•8800. Request to be placed on the mailing list for future editions of business and technology white papers. •



Taligent

10725 N. De Anza Boulevard
Cupertino, CA 95014-2000
408 255 2525

© Taligent, Inc. 1993, all rights reserved.
The Taligent identifier is a trademark of
Taligent, Inc. Produced in the USA.

TT1/1/93